Electronic Notes in Theoretical Computer Science 63 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume63.html 16 pages

Portable Resource Control in Java: Application to Mobile Agent Security

Walter Binder [a,1], Jarle G. Hulaas [b] and Alex Villazón [b,2]

[a] *CoCo Software Engineering, Margaretenstr. 22/9, 1040 Vienna, Austria*
Email: w.binder@coco.co.at

[b] *University of Geneva, rue Général Dufour 24, 1211 Geneva 4, Switzerland*
Email: Jarle.Hulaas@cui.unige.ch

Abstract

Prevention of denial-of-service attacks is indispensable for distributed agent systems to execute securely. To implement the required defense mechanisms, it is necessary to have support for resource control, i.e., accounting and limiting the consumption of resources like CPU, memory, and threads. Java is the predominant implementation language for mobile agent systems, even though resource control is a missing feature on standard Java platforms. Moreover, prevailing approaches to resource control in Java require substantial support from native code libraries, which is a serious disadvantage with respect to portability, since it prevents the deployment of applications on large-scale heterogeneous networks. This article describes the model and implementation mechanisms underlying the new resource-aware version of the J-SEAL2 mobile agent kernel. The resource control model is based on a set of requirements, where portability is very significant, as well as a natural integration with the existing programming model. The implementation consists of a combination of Java byte-code rewriting with well-chosen enhancements in the J-SEAL2 kernel. Realization of a resource control system may be prompted by motivations such as the need for application service providers to guarantee a certain quality of service, or to create the support for usage-based billing. In this article the design strategy is however focussed on security, and more specifically on preventing denial-of-service attacks originating from mobile agents running on the platform. Initial performance measurements are also presented, which back our approach.

1 Introduction

Java [15] was designed as a general-purpose programming language, with special emphasis on portability in order to enhance the support of distributed

[1] Supported by CoCo Software Engineering GmbH.
[2] Supported by the Swiss National Science Foundation.

applications. Therefore, it is natural that access to low-level, highly machine-dependent mechanisms were not incorporated from the beginning. New classes of applications are however being conceived, which rely on the facilities offered by Java, and which at the same time push and uncover the limits of the language. These novel applications, based on the possibilities introduced by code mobility, open up traditional environments, move arbitrarily from machine to machine, execute concurrently, and compete for resources on devices where everything from modest to plentiful configurations can be found. We are therefore witnessing increased requirements regarding fairness and security, and it becomes indispensable to acquire a better understanding and grasp of low-level issues such as resource management.

Operating system kernels provide mechanisms to enforce resource limits for processes. The scheduler assigns processes to CPUs reflecting process priorities. Furthermore, only the kernel has access to all memory resources. Processes have to allocate memory regions from the kernel, which verifies that memory limits for the processes are not exceeded. Likewise, a mobile agent kernel must prevent denial-of-service attacks, such as agents allocating all available memory. For this purpose, accounting of physical resources (i.e., memory, CPU, network bandwidth, etc.) and logical resources (i.e., number of threads, number of protection domains, etc.) is crucial.

Whereas J-SEAL2 [5,6] is primarily designed for mobile agents, the approach described here is in many ways applicable to other distributed programming paradigms practiced in Java, since the mobile agent paradigm is very comprehensive in terms of involved issues and technologies. The techniques employed in J-SEAL2 could thus greatly improve stability and security in the execution of Java Applets, Servlets, Enterprise JavaBeans, or traditional distributed applications, where strong protection domains and resource control mechanisms are often needed.

The great value of resource control is that it is not restricted to serve as a base for implementing security mechanisms. Application service providers may e.g. need to guarantee a certain quality of service, or to create the support for usage-based billing, in order to amortize investments in hardware and software set at customers' disposal. The basic kernel extensions described here will be necessary to schedule the quality of service or to support the higher-level accounting system, which will bill the clients for consumed computing resources. This article is however restricted to the kernel extensions that were necessary to add resource control to J-SEAL2; faithful to the micro-kernel approach, J-SEAL2 relegates to the higher levels the mechanisms which do not absolutely have to be part of the kernel.

This article is organized as follows. The next section presents the design goals and the resulting resource control model. Section 3 gives an overview of our implementation techniques, for which section 4 presents initial performance measurements. Section 5 compares with related work, whereas section 6 gives a glimpse on future investigations and concludes the article.

2 Objectives and Resulting Model

The ultimate objective of this work is to enable the creation of execution platforms where anonymous agents, or more general programs, may securely coexist without harming each other, and without harming their environment. Examples of such platforms are user-extensible databases [14] or decentralized e-commerce and trading systems as e.g. in [16]. The desire to deploy this kind of platforms translates into the following requirements:

- Sufficiently abstract concepts, in order to make mapping of policies into implementations more straightforward, and with a view to making resource control and eventual billing more manageable.
- Accounting of low-level, physical resources as well as higher-level, logical resources, such as threads.
- Prevention against denial-of-service attacks, which are based on CPU, memory, or communication misuse.
- Fair distribution of resources among concurrent domains, even outside the context of malicious activities.
- Fine-grained load-balancing of mobile agent applications on a cluster of machines.

Since some aspects of resource control are to be manageable by the application developer, it is important that the general model integrates well with the existing J-SEAL2 programming model [5]. The J-SEAL2 kernel manages a tree hierarchy of nested protection domains [3], the so-called *seals*. This model of hierarchically organized protection domains stems from the JavaSeal mobile agent kernel [9]. Protection domains encapsulate mobile agents as well as service components [4]. The J-SEAL2 kernel ensures that protection domains are completely isolated from each other, there is no direct sharing between distinct domains. Furthermore, a parent domain may terminate its children at any time, forcing the children to release all allocated resources immediately.

The resource control facilities shall reflect the hierarchical system structure. Hierarchical process models have been used successfully by operating system kernels, such as the Fluke micro-kernel [12]. The Fluke kernel employs a hierarchical scheduling protocol, CPU Inheritance Scheduling [13], in order to enforce scheduling policies. In this model, a parent domain donates a certain percentage of its own CPU resources to a child process. Initially, the root of the hierarchy possesses all CPU resources.

A general model for hierarchical resource control, like Quantum [18], fits very well to the J-SEAL2 hierarchical domain model. At system startup the

[3] In this article the term 'protection domain' refers to the concept of a *process* or *task* in an operating system, and not to the JDK class `java.security.ProtectionDomain`.

[4] In J-SEAL2 mobile agents are not allowed to directly use certain functionality of the JDK, such as file or network IO, but they have to access dedicated J-SEAL2 services that are executing in separate protection domains.

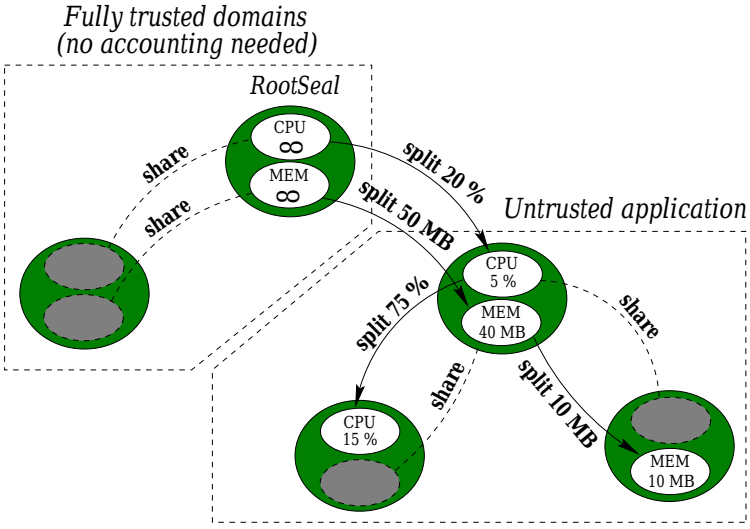

Fig. 1. Illustration of the general resource control model.

root domain, *RootSeal*, owns by default all resources, for example 100% CPU, the entire virtual memory, unlimited network usage, the maximum number of threads the underlying Java Virtual Machine (JVM) [17] is able to cope with, an unlimited number of subdomains, etc. Moreover, the root domain, along with the other domains loaded at platform startup, are considered as completely safe, and, consequently, no resource accounting will be enforced on them. This default behavior may however easily be overridden, if specific configurations should require accounting even for trusted domains.

When a nested protection domain is created, the creator donates some part of its own resources to the new domain. Figure 1 illustrates the way resources are either shared or distributed inside a seal hierarchy. In the formal model of J-SEAL2, the Seal Calculus [25], the parent seal supervises all its subdomains, and inter-domain communication management was the main concern so far. Likewise, in the resource control model proposed here, the parent seal is responsible for the resource allocation with its subseals. This produces a nested structure, where the parent seal is initially the sole owner of its resources, and it may either share them or dispatch fractions of them to its subseals. However, the sum of all resources within a protection domain, e.g., in the *Untrusted application* of Figure 1, remains constant.

Our resource control model stems from further design goals, such as portability and transparency: the next subsections are dedicated to describing these.

2.1 Portability and Transparency

Portability is crucial for the success of any mobile agent platform. There are already some Java-based systems offering resource control facilities, such as Alta [24], GVM [4], KaffeOS [1,2], etc. However, they rely on modified Java runtime systems, which are not portable. As a result, these systems are not suited for large-scale applications that have to support a wide variety of

different hardware platforms and operating systems. Our goal is to provide a general purpose model which is not dependent on specific implementation techniques, and to explore primarily completely portable solutions. This entails that we have to cope with certain restrictions and with performance levels sometimes inferior to those of existing realizations. Our portable approach will nevertheless show its advantages in the longer term: our solution will always perform somewhat slower than the fastest JVMs without resource control mechanisms, but, on the other hand, we will be able to exploit the latest techniques in Java implementation optimizations, which will often not be possible with non-portable implementations.

2.2 Minimal Overhead for Trusted Domains

Since J-SEAL2 is designed for large-scale applications, where a large number of services and agents are executing concurrently, design and implementation must minimize the overhead of resource accounting. Some domains, such as core services, are fully trusted. Their resource consumption need not be controlled by the kernel.

2.3 Support for Resource Sharing

In certain situations protection domains that are neighbours in the hierarchy may choose to share some resources. In this case, resource limits are enforced together for a set of protection domains. As a result, resource fragmentation is minimized. For example, consider an agent creating a subagent for a certain task. Frequently, the creating agent does not want to donate some resources to the subagent, but it rather prefers to share its own resources with the subagent. A property of our approach is that if a domain has unlimited access to a resource, this means that it is sharing it with RootSeal.

2.4 Managed Resources

Within each untrusted protection domain, the J-SEAL2 kernel shall account for the following resources (for details, see [7]):

- CPU_RELATIVE defines the relative share of CPU, and is expressed as a fraction of the parent domain's own relative share. In our current implementation, this resource is controlled by periodic sampling of the amount of executed byte-code instructions. The precision of the measurement is moreover implementation dependent [5].
- MEM_ACTIVE is the highest amount of volatile memory that a protection domain is allowed to use at any given moment.

[5] There is a bias, because in the implementation, the reference value is the aggregated consumption measured among untrusted domains and not, as would be expected, the resource taken as a whole.

- THREADS_ACTIVE specifies the maximal number of active threads by protection domain at any moment. Uncontrolled creation of threads has to be avoided, as it results in increased load for the scheduler, and may even crash the JVM.
- THREADS_TOTAL limits the number of threads that may be created throughout the lifetime of a protection domain, as thread creation is an expensive (kernel-level) operation.
- DOMAINS_ACTIVE specifies the maximal number of active subdomains a protection domain is allowed to have at any given moment. This limit exists in order to control management overhead inside the kernel by controlling the complexity of the seal hierarchy at any time.
- DOMAINS_TOTAL bounds the number of subdomains that a protection domain may generate throughout its lifetime, as domain creation and termination are expensive kernel operations.

Note that the kernel of J-SEAL2 is not responsible for network control. This is because the micro-kernel does not provide access to the network. Instead, network access can be provided by multiple services. These network services or some mediation layers in the hierarchy are responsible for network accounting according to application-specific security policies. Let us stress that the network is not a special case, since J-SEAL2, thanks to its homogeneous model, may limit communication with any services, like networking, file IO, etc. In J-SEAL2 resource control for service access (e.g., bandwidth limitations), as well as revocation of service access are handled by the communication model, which is not covered by this article.

Another kind of resource that could be expected in the above list of kernel-managed resources is the total amount of CPU allocated to a given protection domain throughout its lifetime. It is however not clear what the unit of measurement should be for this resource, while still preserving a completely hardware-independent model. Since the main objective of this kind of resource accounting would be to prevent applications from indefinitely cluttering up platforms, in a heterogeneous set of servers it gives more sense to express total life time abstractly as the wall clock time elapsed since the application was started than as the number of consumed CPU cycles. Using the amount of executed Java byte-codes, although portable, was also regarded as too low-level. Measuring wall clock time can be achieved at the application level, by establishment of a controller agent with sufficient rights to kill all misbehaving applications. In J-SEAL2, when a parent disposes of a child seal, all resources are guaranteed to be freed properly, and accounting of total CPU time was therefore discarded from the kernel.

Finally, there is also no such resource as MEM_TOTAL, a limit to the accumulated amount of memory used throughout the lifetime of a protection domain. It could be needed to prevent the kind of denial-of-service attacks where a malicious agent creates a lot of dynamic objects in order to keep

the CPU busy with garbage collection, but its implementation would require maintenance of an additional counter, which we preferred to avoid. Instead, J-SEAL2 will take preventive action by charging an abstract amount of CPU as a compensation for the garbage collection induced by each object created.

3 Implementation

In this section we sketch the techniques we are resorting to for a completely portable implementation of the resource control model discussed in the previous section. Implementation details can be found in [7]. Since accounting for logical resources, such as threads and subdomains, requires only simple modifications to a few J-SEAL2 kernel primitives, we focus on accounting for physical resources, such as memory and CPU.

3.1 No Direct Sharing

Since its initial release the J-SEAL2 kernel is designed to ease the integration of resource control facilities. It guarantees accountability, i.e., each object belongs to exactly one protection domain. References to an object exist only within a single domain, i.e., in J-SEAL2 there is no direct sharing of object references between distinct domains. Therefore, it is possible to account each allocated object to exactly one protection domain. This feature not only simplifies resource accounting, but it is also crucial for immediate resource reclamation during domain termination.

3.2 Byte-code Rewriting

In our approach we employ byte-code rewriting techniques both for memory and CPU accounting. This is because it is to our understanding the only entirely portable way to implement the needed accounting mechanisms. It is unrealistic to expect the source code of every application to be available for modification. Moreover, if we want guarantees against denial-of-service attacks, we cannot rely on foreign code to perform any voluntary self-limiting operations, whereas if we modify its byte-code before it starts executing, we can 'oblige' it to provide any information needed by the kernel and to obey any restriction imposed on it by the environment.

In our implementation the byte-code of a Java class is modified before it is loaded and linked by the JVM (for details about class-loading, see [17]). Code for memory accounting is inserted before each memory allocation instruction. CPU accounting uses an abstract measure, the number of executed byte-code instructions. Therefore, code for CPU accounting is included in every basic block of code. Rewriting for memory accounting is done before rewriting for CPU accounting, because memory accounting inserts additional byte-code instructions to enforce memory limits, while accounting CPU consumption does not involve any object allocation.

In our implementation a thread executing in a domain requiring memory or CPU control has associated accounting objects representing resource limit and current consumption. Because access to the accounting objects may be extremely frequent, we are rewriting non-native methods in order to pass the necessary accounting objects as additional arguments. Native methods are excluded from rewriting, because we cannot account for memory allocated and CPU time consumed by native code. We are relying on modern inter-modular register allocation algorithms implemented by state-of-the-art JVMs to minimize the overhead of passing the accounting objects.

3.3 Class-loading

The J-SEAL2 kernel distinguishes between shared and replicated classes [6]. Shared classes are loaded by the system class-loader (they exist only once in the JVM), while replicated classes, such as the classes of a mobile agent, are loaded by the class-loader of a protection domain (they are reloaded in each domain). All JDK classes [6] as well as most classes from the J-SEAL2 kernel are shared. Certain J-SEAL2 library classes that are frequently used may be shared as well, in order to avoid the overhead of reloading them multiple times.

Since a shared class may be referenced by fully trusted domains (no accounting necessary) as well as by untrusted domains (resource control required), it has to provide multiple different versions of each method. The version without resource control corresponds to the unmodified code, while the version used by untrusted domains takes the accounting objects as extra arguments and includes the necessary accounting instructions.

Shared classes are rewritten off-line (e.g., during the installation of the J-SEAL2 platform), because we cannot modify the system class-loader, which is part of the Java runtime system, in a portable way. Replicated classes are rewritten on-line, immediately before they are linked into the JVM. Therefore, a mobile agent may execute unmodified code on a J-SEAL2 platform where it is trusted, while on another J-SEAL2 installation the code of the same agent may be rewritten for resource control.

3.4 Memory Control

Like in JRes [10], code for memory control is inserted before each memory allocation instruction. The J-SEAL2 kernel maintains weak references to allocated objects in order to detect when an object is reclaimed. Enforcing memory limits requires exact pre-accounting for memory resources, i.e., an overuse exception is raised before a thread can exceed the memory limit of the domain it is executing in. In contrast to JRes, which controls a separate memory limit for each thread, J-SEAL2 is able to enforce a single memory

[6] It is not possible to load a JDK class with a loader different from the system class-loader.

limit for a multithreaded domain, or even for a set of domains in the case of resource sharing.

3.5 CPU Control

CPU accounting is based on an abstract measure, the number of executed byte-code instructions. Basically, instructions for CPU accounting are inserted at the begin of each basic block of code [7]. In order to prevent competition for a common CPU account object, every thread executing in an untrusted domain maintains its own CPU account. A high-priority scheduler thread executes periodically in order to ensure that assigned CPU limits are respected. It is responsible for accumulating the accounting data of all threads executing in a domain. The scheduler thread compares the number of executed byte-codes with the desired schedule and adapts the JVM thread priorities of individual threads in order to control the CPU consumption of different protection domains [8].

3.6 Accounting for Garbage Collection

In order to prevent denial-of-service attacks by causing the garbage collector to consume a considerable amount of CPU time (e.g., an attacker may create a lot of garbage without exceeding its memory limit), the J-SEAL2 kernel has to account for the time spent by the garbage collector. Since the exact CPU time spent by the garbage collector is not known, we are using an abstract measure. The J-SEAL2 administrator defines a rough approximation of the number of byte-code instructions required to reclaim an object.

Before an object is allocated, the J-SEAL2 kernel charges the CPU account of an allocating thread. That is, a domain has to 'pay' for the garbage it eventually will produce at the time it 'buys' an object. This simple approach has the advantage that a domain is charged for all garbage it produces, even if the domain has already terminated when some objects are reclaimed.

3.7 Avoiding Native Code

With the aid of byte-code rewriting techniques, it is not possible to account for memory allocation and CPU consumption in native code. Untrusted applications are not allowed to bring native code libraries into the system. Concerning JVM-provided standard operations, the J-SEAL2 kernel tries to compensate for resources used by native code and prevents untrusted domains from using certain functionality leading to a significant resource consumption by native

[7] In [7] we present some simple optimization rules allowing to combine accounting code for a set of basic blocks.

[8] In J-SEAL2 threads running in untrusted domains are not allowed to modify their own priorities. We are using extended byte-code verification to ensure this property (for details, see [6]).

code. In the following we describe some important cases of resource consumption in native code and how J-SEAL2 solves them:

- Class-loading: The Java runtime system manages an internal table of loaded classes. Memory for compiled methods is allocated by the Just-in-Time compiler, which is usually implemented in native code. However, the set of classes untrusted domains (e.g., mobile agents) are allowed to access is limited and known to the J-SEAL2 kernel. Therefore, the kernel accounts for the classes using an approximation, which is proportional to the size of the class-files.
- Deserialization: J-SEAL2 uses Java serialization in order to create messages to be transferred across domain boundaries. When the receiving domain opens a message, it is being deserialized using the class-loader of the receiving domain to resolve class names. Deserialization requires native methods to allocate objects without invoking their constructors. J-SEAL2 solves this hurdle by storing the amount of objects for each type, which is part of the serialized object graph, in the message. The receiver performs resource checks before deserializing the message.
- Object cloning: Java supports object cloning to create shallow copies of objects. The shallow copy is allocated by a native method. A simple solution is to forbid untrusted domains to use object cloning.
- Reflection: The Java reflection API provides a mechanism to indirectly create a new instance of a class. The object is allocated by native code. J-SEAL2 simply prevents untrusted domains from using the reflection API.

4 Evaluation

While in J-SEAL2 the overhead for memory control is comparable to the overhead caused by JRes [9] [10], the overhead of CPU control based on byte-code rewriting techniques has to be examined carefully, because such an approach has not been used before. In this section we present some initial performance measurements proofing that the overhead due to our completely portable implementation of CPU accounting is acceptable on modern JVM implementations [10].

We measured two well-known micro-benchmarks, *Fib* – the recursive algorithm for the calculation of fibonacci numbers (used to compute the 35th fibonacci number), and *Sort* – the iterative bubble-sort algorithm (used to sort an array of 10000 integer values in ascending order) [11].

[9] For an application allocating a new object every 250 byte-code instructions, the overhead for memory control is less than 18%, if no memory limits are exceeded.

[10] We are not measuring the overhead for CPU control incurred by the scheduler, as it can always be kept small by choosing an appropriate time-slice.

[11] With the aid of our byte-code rewriting tool, we also measured the accounting overhead for the SPEC JVM98 benchmarks. Unfortunately, due to space limitations, we are not able

Table 1 summarizes our measurements, which were collected on a Windows NT 4.0 workstation (Intel Pentium II, 400MHz clock rate) with 5 different JVM implementations. In order to minimize the impact of compilation and garbage collection, all results represent the median of 5 different measurements. For each measurement, table 1 shows the execution time of the micro-benchmark in milliseconds, as well as the speedup of the original code compared to the rewritten version. We measured code rewritten without any optimizations, as well as the code resulting from the application of some simple optimization rules that help to minimize the accounting overhead (for details, see [7]). For each JVM implementation, we applied the set of optimizations that gave the best results.

Table 1
Overhead for CPU accounting (time in milliseconds).

		Sun JDK 1.3				Sun JDK 1.2.2				IBM JDK 1.3	
		Classic (Interpreter)		Hotspot Client		Classic (JIT)		Hotspot Server 2.0		Classic (JIT)	
Fib	original	13029	(1,00)	1542	(1,00)	1031	(1,00)	1032	(1,00)	991	(1,00)
	rewritten	22933	(1,76)	2123	(1,38)	1773	(1,72)	1502	(1,46)	1522	(1,54)
	optimized	16825	(1,29)	1732	(1,12)	1432	(1,39)	1232	(1,19)	1131	(1,14)
Sort	original	16954	(1,00)	1812	(1,00)	1212	(1,00)	1352	(1,00)	782	(1,00)
	rewritten	44344	(2,62)	2774	(1,53)	2434	(2,01)	2564	(1,90)	2323	(2,97)
	optimized	34650	(2,04)	2423	(1,34)	1513	(1,25)	2143	(1,59)	1543	(1,97)

Our measurements for the recursive fibonacci method show that our optimizations allow to reduce the accounting overhead to 12–19% on modern JVM implementations, such as Sun's Hotspot VMs and IBM's Classic VM. These results also indicate that the overhead of passing the additional method argument (the CPU account) is reasonably small. The iterative bubble-sort benchmark incurs more overhead, because the accounting instructions represent the major part of the rewritten code and cannot be removed from the loops, i.e., the results for the bubble-sort benchmark show the worst-case of deterioration of performance to be expected in real applications.

5 Related Work

We distinguish two categories of related work on adding resource control to Java: those which have security as main objective, and those which follow other motivations.

to present all results in this article. However, the values given in table 1 are representative for real applications.

5.1 Resource Control for Security Purposes

Compared to existing proposals for realizing resource control in Java, we broadly differentiate our approach in two ways: first, whether the model supports a process-based approach, with well-defined domain boundaries and resource allocation for each application, and, second, to which extent the implementation is portable or not.

JRes [10] is a resource control system which takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads; in other words, there is no notion of application as a group of threads, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource accounting system and does not enforce any separation of domains; covering this other aspect is the goal of J-Kernel [26], a complementary project of the same research team. For its implementation, JRes does not need any modification to the JVM, but relies on a combination of byte-code rewriting and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying OS, which requires native code to be accessed [12]. For memory accounting, it essentially uses byte-code rewriting, but still needs the support of a native method to account for memory occupied by array objects.

KaffeOS [1] is a Java runtime system which supports the OS abstraction of *process* to isolate applications from each other, as if they were run on their own JVM. Thanks to KaffeOS, a modified version of the freely available Kaffe virtual machine [27], it is possible to achieve resource control with a higher precision than what is possible with byte-code rewriting techniques, where e.g. memory accounting is limited to controlling the respective amounts consumed in the common heap, and where CPU control does not account for time spent by the common garbage collector working for the respective applications. The KaffeOS approach should by design result in better performance, but is however inherently non-portable. This means that optimizations found in compilers and standard JVMs are not benefited from: in a recent publication [2] the authors report that, in absence of denial-of-service attack, IBM's compiler and JVM [20] is 2–5 times faster than theirs.

Developed by the same team as KaffeOS, Alta [24] is a prototype based on the Fluke hierarchical process model, and implemented on the Kaffe virtual machine. The main differences with KaffeOS are that a single garbage collector is responsible for all applications, and that Alta entirely respects the hierarchical process model of Fluke by providing resource control APIs, whereas KaffeOS only retains a more implicit nested CPU and memory management scheme.

NOMADS [22] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks.

[12] More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.

The NOMADS execution environment is based on a Java compatible VM, the Aroma VM, a copy of which is instantiated for each agent. There is no resource control model or API in NOMADS; resources are managed manually, on a per-agent basis or using a non-hierarchical notion of group. Relying on a specialized VM, it follows that the overhead is smaller than with our approach; currently, CPU control is however not implemented.

Many other systems are proposed in the literature, but none of them are as complete as JRes, Alta, KaffeOS, and NOMADS. An excellent recent overview is provided in [3]. To summarize, we might say that J-SEAL2 proposes a protection model inspired both from Alta and J-Kernel, and a memory accounting implementation that is more reminiscent of JRes.

5.2 Other Java-centric Approaches to Resource Control

There are several lines of research, where environments and analysis tools have been designed that can be exploited more or less with the same objectives as exposed in this article.

The Real-Time for Java Experts Group [8] has published a proposal to add real-time extensions to Java. One important focus of this work is to ensure predictable garbage collection characteristics in order to meet real-time guarantees. For instance, the specification provides for *scoped memory areas* with a limited lifetime, which could be implemented – or at least simulated – with the J-SEAL2 extensions described in the present article. Another real-time system, PERC [19], extends Java to support real-time performance guarantees. To this end, the PERC system analyzes Java byte-codes to determine memory requirements and maximal execution times, and feeds that information to a real-time scheduler. The objective of real-time systems is to provide precise guarantees e.g. for worst-time execution; our focus, on the other hand, is on computing approximated resource consumptions in order to prevent denial-of-service attacks. We are more interested in the relative values of applications, and less in absolute figures. This is confirmed by the fact that we are not trying to estimate their real CPU consumption, but rather to compare the respective number of executed byte-codes.

Profilers constitute another class of tools that have many things in common with resource control: both intend to gather information about resource usage. Profilers however are designed to help developers optimize the efficiency of their applications, and not to externally control their resource consumption. The Java Virtual Machine Profiling Interface (JVMPI) [21] is an API created by Sun; it is a set of hooks to the JVM which signals interesting events like thread start and object allocations. Java Usage Monitor (JUM) [11] is a tool which builds upon JVMPI to help the developer determining how much CPU is consumed by the different threads of an application and how much memory they use. JUM needs native code to obtain information from the underlying OS about how CPU time is allocated, and is therefore not portable.

Interestingly, JUM is able to also account for objects allocated by native code. However, JUM is not able to enforce memory limits. While J-SEAL2 allows for exact pre-accounting of memory resources, where an overuse exception is generated before a thread exceeds its memory limit, a resource control mechanism based on JUM can only react after a memory overuse is detected. In addition to these limitations, JVMPI is an experimental interface, it is not yet a standard profiling interface.

Finally, we mention some approaches that rely on economics-based theories, using virtual currencies to achieve natural load-balancing of concurrent applications, as well as recycling of unused resources in open and distributed environments, with the anticipated side-effect of preventing denial-of-service attacks [23]. Our focus is however more on how to implement the basic resource accounting mechanisms on a specific platform, Java, than on the design of high-level – and distributed – resource allocation policies. Nevertheless, whereas the spirit of this article is rather conservative, it does not exclude the application of the presently described techniques to the implementation of open computational markets.

6 Current Status and Conclusion

While the first version of the byte-code rewriting tool is finished, the integration of full resource control into the J-SEAL2 kernel is still in progress. Also on our immediate todo-list is the development of high-level programming tools in order to support a friendlier event notification mechanism than the overuse exceptions generated by the J-SEAL2 kernel. User-specified thresholds should enable applications to receive warnings in a timely manner before the actual overuse happens.

Whereas other approaches focus on high performance, or demonstrate a long-term, deep re-design of the Java runtime system, our proposal might be grossly characterized as a language-based patch. Our resource control system does indeed not provide the same level of accuracy of measurements and execution speed. On the other hand, J-SEAL2 fulfills its job of isolating applications from each other, and particularly of preventing denial-of-service attacks originating from inside the execution platform. Moreover, the complete compatibility and portability of our approach makes it immediately usable for the benefit of large-scale distributed agent systems, especially when mobile code is involved.

Acknowledgements

The authors would like to acknowledge the helpful contributions of Rory G. Vidal. Additional credits go to Julien Francioli, Rudolf Freund, Andreas Krall, Patrik Mihailescu, Klaus Rapf, and Jan Vitek for their valuable support.

References

[1] G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, Mar. 1999.

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[5] W. Binder. J-SEAL2 – A secure high-performance mobile agent system. In *IAT'99 Workshop on Agents in Electronic Commerce*, Hong Kong, Dec. 1999.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.

[7] W. Binder, J. Hulaas, and A. Villazón. Resource control in J-SEAL2. Technical Report Cahier du CUI No. 124, University of Geneva, Oct. 2000. `ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf`.

[8] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA, 2000.

[9] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.

[10] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 21–35, New York, USA, Oct. 18–22 1998. ACM Press.

[11] F.-X. Le Louarn. JUM, a Java Usage Monitor. Web pages at `http://www.iro.umontreal.ca/~lelouarn/jum.html`.

[12] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 101–116, Berkeley, CA, USA, Feb. 22–25 1999. Usenix Association.

[13] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[14] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and portable database extensibility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 390–401, New York, USA, June 1–4 1998. ACM Press.

[15] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1st edition, 1996.

[16] J. Hulaas, L. Gannoune, J. Francioli, S. Chachkov, F. Schütz, and J. Harms. Electronic commerce of internet domain names using mobile agents. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, Nashville, TN, USA, Oct. 1999.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[18] L. Moreau and C. Queinnec. Design and semantics of Quantum: a language to control resource consumption in distributed computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, CA, USA, Oct. 1997.

[19] K. Nilsen. Java for real-time. *Real-Time Systems Journal*, 11(2), 1996.

[20] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[21] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/index.html`.

[22] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In C. Sierra, G. Maria, and J. S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 163–164, NY, June 3–7 2000. ACM Press.

[23] C. F. Tschudin. Open resource allocation for mobile code. In *Proceedings of The First Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[24] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[25] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[26] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. J-Kernel: A capability-based operating system for Java. *Lecture Notes in Computer Science*, 1603:369–394, 1999.

[27] T. Wilkinson. Kaffe - a Java virtual machine. Web pages at `http://www.kaffe.org/`.